

J-Viz: Sibling-First Recursive Graph Drawing for Visualizing Java Bytecode

Md. Jawaherul Alam, Michael T. Goodrich, and Timothy Johnson

Dept. of Computer Science, University of California, Irvine, CA USA

Abstract. We describe a graph visualization tool for visualizing Java bytecode. Our tool, which we call J-Viz, visualizes connected directed graphs according to a canonical node ordering, which we call the *sibling-first recursive* (SFR) numbering. The particular graphs we consider are derived from applying Shiver’s k -CFA framework to Java bytecode, and our visualizer includes helpful links between the nodes of an input graph and the Java bytecode that produced it, as well as a decompiled version of that Java bytecode. We show through several case studies that the canonical drawing paradigm used in J-Viz is effective for identifying potential security vulnerabilities and repeated use of the same code in Java applications.

1 Introduction

The Space/Time Analysis for Cybersecurity (STAC) program [10] at the U.S. Defense Advanced Research Projects Agency (DARPA) aims to develop new program analysis techniques and tools for identifying vulnerabilities related to the space and time resource usage behavior of algorithms, and specifically to vulnerabilities based on algorithmic complexity and side channel attacks. STAC seeks to enable security analysts to identify algorithmic resource usage vulnerabilities in software to support a methodical search for them in the software upon which the U.S. government, military, and economy depend [10].

Our Contributions. In this paper, we describe a tool, the JVM abstracting abstract machine (Jaam) Visualizer, or “J-Viz” for short, which is intended for use by security analysts to perform such searches through the exploration of graphs derived from Java bytecode. We are not attempting to solve the problem of identifying software algorithmic complexity attacks in a completely automated way, but instead we are providing a means for doing semi-automated analysis that increases the efficiency of a human analyst. The workflow for our tool involves taking a given program, specified in Java bytecode, and constructing a control-flow graph of the possible execution paths for this software, using a framework known as *control flow analysis* (CFA) [22]. Our tool then provides a human analyst with an interactive view of this graph, including heuristics for aiding the identification of which parts of the provided program seem suspicious.

One of the main components of our J-Viz tool involves visualizing control-flow graphs in a canonical way based on a novel vertex numbering scheme that we call the *sibling-first recursive* numbering. This numbering scheme is essentially a hybrid between the well-known breadth-first and depth-first numbering schemes, but differs from both in a way that appears to be more useful for visualizing control-flow graphs. In particular, as we show in some case studies, this approach tends to highlight areas in software where code is repeated and it also allows us to provide visual highlights of code that is contained in deeply nested loops. We designed J-Viz with the following goals in mind:

- We want users to be easily able to recognize patterns in source code from our visualizations. Thus, similar sections of code should produce similar subgraphs, which should be drawn in a similar way.
- We want to use a hierarchical visualization, in which users can collapse or expand sections of the graph to different levels of detail. But we also want them to be able to build a consistent mental model of the graph. Thus, drawings should not drastically shift the relative positions of the vertices when sections are collapsed or expanded.
- No matter what sequence of actions the user performs, drawings should be consistent. That is, the same view of a graph, in which the same set of nodes are collapsed and expanded, should always be drawn in the same way.
- Our system should rank sections of the graph by how likely they are to produce vulnerabilities, and display this information visually to the user.

We believe that J-Viz makes substantial progress in achieving these goals, and we provide several case studies in this paper that support this conclusion.

Related Work. Although it is using different means to achieve mental map preservation, the J-Viz system follows in a long line of research on techniques directed at preserving the mental map of a graph drawn dynamically. For instance, Misue *et al.* [20] discuss node movement adjustments, including avoiding node overlaps, for preserving the mental map. Diehl and Carsten [7] discuss force-directed approaches for preserving the mental map between instances of a changing graph. Goodrich and Pszona [12] study efficient algorithms for minimizing vertex movements as a graph is incrementally revealed in an online manner. With respect to existing software systems, the Graphviz [8] and GraphAEL [9] systems both include algorithms intended to preserve the mental map as a graph is modified. Bridgeman and Tamassia [3] formally study metrics for characterizing mental map preservation between different instances of a changing graph. So as to provide an empirical basis for such work, a user study of Purchase *et al.* [21] supports the thesis that preserving the mental map for graph visualization is a useful goal to aid users in performing tasks on graphs.

Visualization tools have also previously been applied to source code. Doxygen [14], a tool for automatically generating documentation, can produce

various kinds of diagrams for visualizing code, including call graphs. It is generally configured to use the *dot* [11] tool from GraphViz to draw these graphs hierarchically. Similarly, Visual Studio can visualize call graphs to aid programmers in debugging applications [6]. In contrast with these systems, our J-Viz tool provides four main features that these tools do not provide. First, J-Viz shows a greater level of detail, since it analyzes code at the level of individual instructions rather than methods. Second, J-Viz allows the user to interact with a graph and produce multiple views of the same Java bytecode. Third, the layout algorithm used in J-Viz is designed to draw similar code fragments in the same (canonical) way, so as to highlight portions of repeated code. Fourth, J-Viz guides the user to potential security vulnerabilities, by highlighting nodes that are believed to be risky based on algorithmic complexity (or other factors), whereas these other systems were not focused on software security.

Another tool, Jinsight [5], can be used to profile a Java program to provide various views of resource usage, such as highlighting which instances of a class have taken the most time or used the most memory. This tool does not provide a full graph of the program’s possible execution paths, however, which we believe to be essential for detecting security vulnerabilities.

2 Graph Generation via Static Analysis

In this section, we review the process that takes Java bytecode as input and produces the graphs that are visualized in J-Viz. These graphs are produced using the *JVM abstracting abstract machine (Jaam)* tool [4] developed at the University of Utah based on the work of Van Horn and Might [17], which itself is based on control-flow analysis (CFA) framework known as *k-CFA* [18, 22]. Because there could be exponentially many possible execution paths of any given program, which would be too large to visualize and reason about, the *k-CFA* framework compresses execution paths into a graph of reasonable size that represents possible executions of a Java program at the instruction level. Such a graph is called *sound* if it represents every possible execution path, and *precise* if it excludes every impossible execution path. The *k-CFA* framework is sound, and it has a tunable degree of precision based on the integer parameter, *k*, albeit at the cost of creating additional states in the graph for larger values of *k*.

At the lowest level of the hierarchy, 0-CFA, we discard contextual information and generate one state, which forms a vertex in the graph representation, for each line of Java bytecode. Then we add edges for every possible state that could be reached from a given state. For example, a return statement will have an edge to every place from which our current method could have been called. At the next level, 1-CFA, each state also tracks the location from which its method was called. (See Fig. 12 in Appendix B for example graphs produced by 1-CFA.) This easily generalizes to higher levels, so that for *k-CFA*, each state stores

the locations of the previous k function calls. This added information allows many of the spurious branches produced by 0-CFA to be pruned. (For additional information, please see more detailed descriptions of k -CFA [17, 18, 22].)

0-CFA is known to take $O(n^3)$ time to construct a graph for a program with n lines of code, and this is believed to be tight [15]. k -CFA is EXPTIME-complete for functional languages [16], but can be solved in polynomial time for object-oriented languages [19]. Thus, to provide a reasonable balance between soundness, precision, and efficiency, the version of the Jaam static analyzer used for the work of this paper is based on 1-CFA. To summarize, then, the Jaam static analyzer takes as input Java bytecode for a given program and produces an directed graph, G , that represents the results of a 1-CFA performed on this bytecode. This graph is *ordered*, in the sense that the outgoing edges for each node are sorted according to the order in which the corresponding instructions appear in the Java bytecode.

3 Our Sibling-First Recursive Layout Algorithm

Our approach to the layout of graphs produced by the Jaam tool [4] is based on what we believe is a novel graph numbering scheme, which we call a *sibling-first recursive* (SFR) numbering. Intuitively, SFR is a hybrid numbering scheme that combines features of a breadth-first search (BFS) numbering and a depth-first search (DFS) numbering. We show in Figure 1 the difference between algorithms for doing a depth-first search (DFS) ordering of a directed graph and a sibling-first recursive (SFR) numbering. See also Figures 7, 8, and 11 in the appendix, which illustrate SFR spanning trees and their differences with DFS and BFS spanning trees.

```

Algorithm DFS( $v$ ):
  Number  $v$  as vertex  $n$ 
  Increment  $n$ 
  for each directed edge  $(v, w)$  do
    if  $w$  is not numbered then
      Identify  $w$  as a child of  $v$ 
      DFS( $w$ )

```

```

Algorithm SFR( $v$ ):
  for each directed edge  $(v, w)$  do
    if  $w$  is not numbered then
      Identify  $w$  as a child of  $v$ 
      Number  $w$  as vertex  $n$ 
      Increment  $n$ 
    for each directed edge  $(v, w)$  do
      if  $w$  is a child of  $v$  then
        SFR( $w$ )

```

Fig. 1. DFS and SFR algorithms to explore the connected component of a vertex, v , in a directed graph, G . We assume there is a global variable, n , which is used to number the vertices. In the case of DFS, we initialize $n = 1$ and call DFS(v) on a vertex v that is to become the root of the DFS tree. In the case of SFR, we initialize a vertex, v , as the root of the SFR tree, numbering it as vertex 1, and we set $n = 2$ and call SFR(v).

Our motivation for using the SFR numbering is that we feel it produces a rooted spanning tree that corresponds more intuitively with the way that programmers conceptualize the main “backbone” of the control flow of their software. For example, it places the true-false branches of if statements as children of the condition that branches to them. In addition, it places the multiple branches of a switch statement as children of the condition that branches to them, even if some of the branches flow-through to other branches. (E.g., see Fig. 2.) Furthermore SFR numbering also enables the viewer to visually identify isomorphic subgraphs of the graph, corresponding to identical, repeated or equivalent lines of code; see Fig. 9 in Appendix A.

High-Level Description of Our Layout Algorithm. At a high level, there are five steps in our algorithm for producing a drawing of the graph, G :

1. We construct an SFR numbering and rooted spanning tree, T , for our input graph, G , which will be used as the “backbone” of our drawing.
2. We draw the tree T using a recursive placement algorithm.
3. We add the edges of G that are not in the tree T .
4. We highlight in our drawing the sections of our graph that are most likely to contain vulnerabilities, based on various criteria.
5. We automatically group subsets of nodes before displaying the entire graph to the user, in a way that allows the user to expand such collapsed nodes.

In the remainder of this section, we describe in more detail each of the above steps in our layout algorithm.

Constructing an SFR search tree. In our first step, we construct a rooted (ordered) SFR spanning tree, T , for the graph, G , produced from the Jaam tool [4]. The algorithm we use to perform this construction is exactly the SFR algorithm shown in Fig. 1, with the added detail that as we traverse the graph G to construct our SFR search tree, T , we process the out-edges from each vertex using the ordering for G , consistent with the intuitive way programmers naturally organize branches for different types of software branch points. As we highlight in one of our case studies, this approach tends to produce almost identical drawings for repeated (e.g., cut-and-pasted) software code fragments. It also produces ordered combinatorial layouts for each of the following types of code constructs, as shown in Fig. 2.

- If-else conditional statement: the true and false components are siblings, with the true component coming first.
- Switch conditional statement: the different branches of the switch statement are siblings of the conditional statement, ordered by their appearance in the code (even if there are non-tree edges between them that would be representing flow-throughs from one branch to another).

- While/for loop: the end-of-loop statement and loop body are both siblings of the conditional statement, with the end-of-loop statement coming first.
- Do-while loop: the start statement, loop body, and conditional statements are in a single path, with a non-tree edge leading back to the start statement.

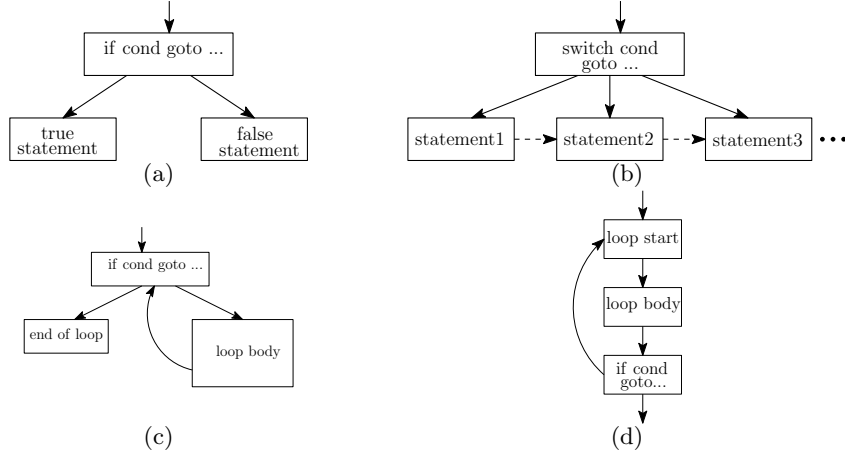


Fig. 2. Sample graph layout for (a) if-else conditional statement, (b) switch statement (dashed edges for flow through statements), (c) for and while loop, (d) do-while loop.

Drawing the Nodes of our SFR Search Tree. Once we have constructed our (ordered) SFR search tree, T , we draw it recursively, starting from the leaves of T . Each parent is drawn on a row above all of its children. Chains of nodes are drawn in a vertical column. When we reach a branch point, we lay out each of the subtrees from left to right. We require that only a direct descendant of a node can be placed directly underneath it. This means that each subtree, no matter its size, will have an entire vertical lane reserved for it from top to bottom in our graph. See Fig. 3.

This requirement might at first seem to waste space in our drawing, but it maintains consistency when a user expands or collapses connected sections of nodes. To see why this is so, suppose that two nodes are placed at the same x -coordinate, but neither is an ancestor of the other. Suppose further that the user then chooses to collapse the set consisting of the path from each of these nodes up to their lowest common ancestor. If this happens, then if we simply shift up that portion of the spanning tree, then we will cause overlaps, which would require shift of nodes to fix. (See Fig. 4.) But such a shift would be moving nodes in a way that could be detrimental to the mental map. Thus, rather than produce a compact drawing that reuses vertical space, we use the scheme described above, which tends to preserve the mental map even as we

would be collapsing or expanding paths in the spanning tree, T , and shifting the remaining portions accordingly.

Drawing Edges. After we have placed the nodes of our ordered spanning tree, T , we must draw all of the edges in our graph. In our case we choose to draw downward edges as straight line segments, and we then draw upward edges as curved segments. In addition to drawing arrows at the ends of such segments, this convention provides a visual cue for which direction an edge is pointing. It also prevents upward edges from lying on top of downward edges. For example, this makes the drawing of the graph of software implementing the bubblesort algorithm, shown in Fig. 10 in Appendix B, more readable.

Highlighting. After the nodes and edges of the graph, G , are placed, we highlight vertices to guide the user on where to begin examining it to discover possible security vulnerabilities. For attacks based on increasing the running time for the software on certain inputs, we want to highlight nodes that are likely to be visited the greatest number of times during an execution. So we color our

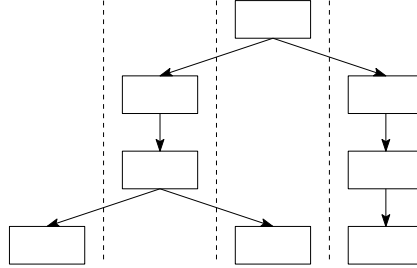


Fig. 3. A tree drawn according to our algorithm. The dashed vertical lines show separation of subtrees into unique lanes.

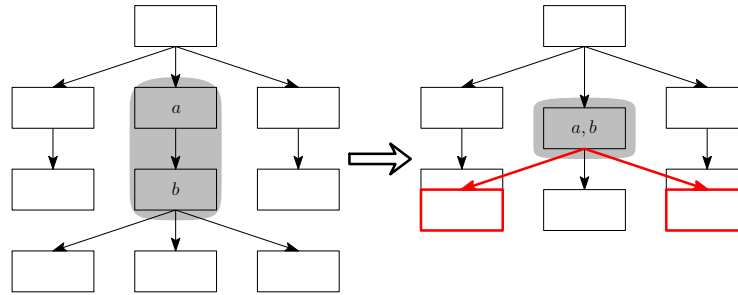


Fig. 4. An example of how breaking our drawing style can cause problems. When the user collapses nodes a and b , some of the nodes need to be shifted. To avoid this problem, we forbid a node from lying directly above a node that is not a direct descendant.

states from green to red based on how likely each node is to be involved in a vulnerability. There are several ways to determine such a vulnerability ranking. We have considered the following three:

- We could perform a recurrence analysis that computes an upper bound on how many times each node is visited. Automated recurrence analysis has progressed to the point of being able to provide strong upper bounds on many simple algorithms [1]. But we do not believe that this is yet possible for programs such as the ones we need to examine, which contain thousands or tens of thousands of lines of code, and have a complex loop structure.
- We could profile our code, by providing a sample input and counting how many times each state is visited. But while this may help for honestly written programs, we believe it is unlikely to identify the kinds of deliberate attacks that we need to discover. The programs we are given should in most cases perform well, because otherwise they would not be used at all. But some may have a hidden trigger that causes them to run for much longer.
- We could count the number of nested loops that contain each node. This is a somewhat naive method, but it does seem to give a reasonable heuristic, as will be seen in our case studies. It is also feasible to compute even for very large graphs. Hence this is the method that we choose to use.

In order to do this highlighting, of course, we need to determine for each node its level of nesting with respect to the loops of the program. We use an adaptation to the SFR tree of a definition by Havlak [13] for DFS trees:

- The outermost loops are the maximal strongly connected components of the directed control-flow graph, G .
- The header for a loop is the first node in the loop that is reached in the SFR tree, T .
- The inner loops are the maximal strongly connected components that remain when the header is removed.

A graph is said to be *reducible* if every cycle has a single entry point [13]. If the graph is reducible, then the loop decomposition does not depend on which rooted spanning is used. But if a cycle has multiple entry points, then the order in which we explore the branches could matter. Thus, in our case, we use the canonical SFR tree, T , that has already been defined for our graph.

To compute loop headers efficiently, we use an algorithm from Tao *et al.* [23]. This traverses the ordered spanning tree and passes loop header information up the tree. While their method could take a long time for artificially complex graphs, it takes linear time for most real-world programs, because the spanning trees for such graphs tend to be reducible or “nearly” reducible.

Grouping Nodes. We have included four different ways in which nodes in the graph, G can be aggregated, and we present the initial drawing of G to the

user based on a pre-defined grouping of certain nodes, with some of these pre-collapsed. First, we choose not to explore nodes that correspond to calls to the Java library, since we do not expect it to contain vulnerabilities. Instead, every such call is collapsed to a single line. This prevents us from creating hundreds of thousands of nodes for the Java library. Nevertheless, the static analyzer, Jaam, needs to do this carefully, so that it can approximate the state of Java library objects and predict their later behavior. For example, any object that is added to an ArrayList or a HashMap can later be taken out. Still, we assume that such an identification is given as an annotation to the input graph, G , since this identification is solely the domain of the Jaam tool. Second, we automatically group each connected set of nodes that belong to the same method. That way, if the user is not interested in the details of a given method, they can collapse it to a single node. Third, we automatically group chains of method nodes that were created in the previous step. Generally, having long chains of nodes taking up a large portion of the screen space hinders the user from seeing the branching structure that they need to find. Finally, we allow the user to select a connected set of nodes and collapse them dynamically, along with providing a comment explaining the purpose of the corresponding section of code.

4 The User Interface

In constructing the user interface for the J-Viz system, we relied in part on a survey study of Basil and Keller [2] for software visualizations. They found that the following criteria were considered “absolutely essential” by a majority of participants:

- Search tools for graphical and/or textual elements
- Source code visualization
- Hierarchical representations
- Use of colors
- Source code browsing
- Navigation across hierarchies
- Easy access from the symbol list of the corresponding source code.

To make our system more user-friendly, we have implemented each of these features. We have already discussed our hierarchy for collapsing and expanding vertices, and our use of colors for highlighting nested loops in our graph. We show the other features in use in the screenshot shown in Fig. 13 in Appendix C, and describe each of them below.

Searching. We have included the following searching capabilities in our graph:

- Find nodes by SFR numbering
- Find incoming and outgoing edges for a given node
- Find nodes whose methods contain a given string
- Find nodes whose instructions contain a given string

Context panel. When nodes are selected by the user, the code for their methods are shown in the left panel, with the lines for each node highlighted. Alternatively, a user can select lines from the left panel, and we highlight the nodes in our graph that correspond to those lines of code. The lines of code for the innermost loops inside each method are also highlighted.

Description panel. The right panel displays detailed descriptions for each of the selected nodes in our graph.

Minimap. A minimap is given in the lower left that always shows our full graph. When we zoom in on part of the graph, it is highlighted on the map, so that the user never loses track of where they are.

5 Case Studies

In this section, we describe some case studies we performed to test the effectiveness of the J-Viz system for visualizing Java bytecode and identifying security vulnerabilities that could be triggered by algorithmic complexity attacks. As input to these case studies, we were provided by DARPA with programs to analyze to test our system, some of which were produced by a “red team” tasked with deliberately creating software that contains vulnerabilities to algorithmic complexity attacks.

Our first case study is for a program for verifying a password, which is meant to be kept secret, and not revealing any information about such a password, including whether it is valid or not. In this case, J-Viz was effective in leading a security analyst to a nested loop that checks each character in the password one at a time. In part, because of the way that the SFR spanning tree lays out conditional branches in an intuitive manner and draws loop edges as curved segments, the analyst was able to notice that the password-checking program exits as soon as it finds the first character that does not match. (See Fig. 14 in Appendix C.) The analyst then correctly identified this as a vulnerability (inserted by the red team), since, by timing multiple executions of the program, an attacker can easily determine how many correct characters of a password that they entered with each attempt. Thus, such an attacker could quickly crack the password by a simple iterative search.

Our second case study is for a program for analyzing and classifying images based on features, such as the number of edges or the amount of each color that is present. This program is around 1,000 lines of Java code, and produces about 3,000 nodes in our graph. The goal of the security analyst in this case was to determine if this program can be made to take much longer than it should (specifically, greater than 18 minutes to analyze a 70 KB image). For most images of that size, the program takes around 6 minutes. But, through the

use of J-Viz, a security analyst was able to create an image that would take over an hour to be analyzed. The key insight for the analyst was to pay attention to the highlighting in our visualizer, which showed the deepest nested loops in dark red. (See Fig. 5.)

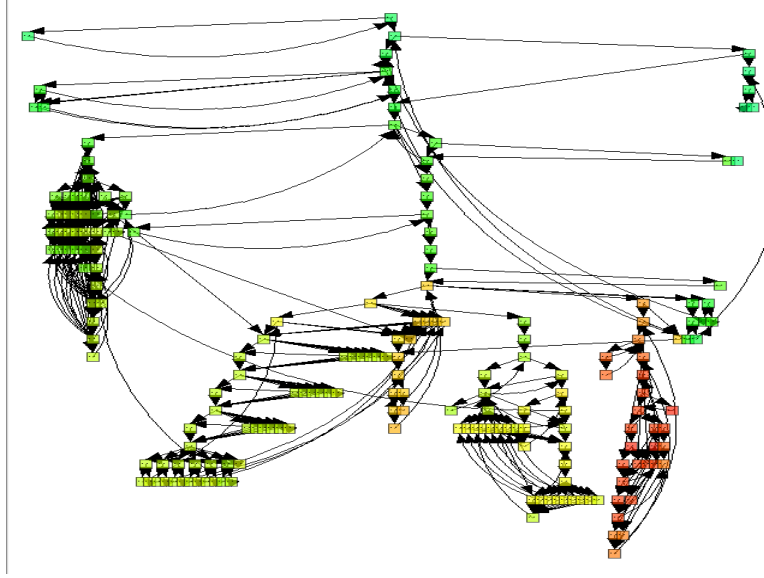


Fig. 5. Our visualization for the graph for an image processor program, showing the red section of a deeply nested loop (which contained a security vulnerability) in the bottom right.

In this case, the analyst noticed that many of these nodes were part of the Mathematics class, which provided custom implementations of various mathematical functions. In particular, the exponential function was implemented using a Taylor series, with the number of terms depending on a function of the RGB value of each pixel. This function was highly non-linear and contained a “spike,” which is shown in Figure 6, which implied a large number of terms in the Taylor series for an image having a particular RGB value. Given this information, the analyst was then able to create an image that triggered this behavior for every pixel, which took over an hour to process, confirming the vulnerability.

In addition to these case studies, we show additional examples and screenshots in Appendix B and C.

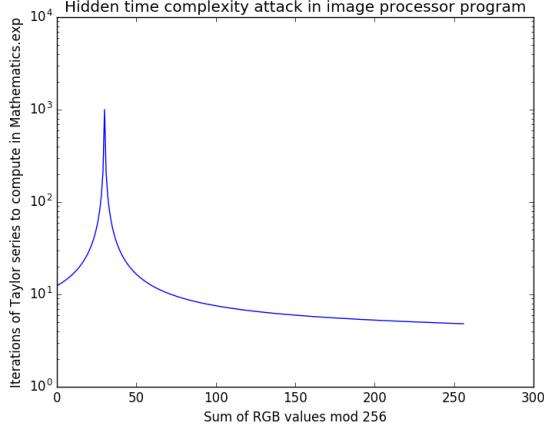


Fig. 6. The time complexity of image processor spikes by two orders of magnitude when the sum of the RGB values is exactly 30.

6 Conclusion and Future work

We have described a new software visualization tool, J-Viz, that uses an SFR numbering scheme for graphs produced using the 1-CFA framework so that similar sections of code should be drawn similarly and deeply nested code portions are placed well and highlighted.

In keeping with recent graph drawing research, we have argued that our system preserves the mental map of the user as they interact with the graph. We also meet all of the criteria that both programmers and researchers have considered essential for software visualization. As a result, our system has already proven to be useful for human analysts in finding various kinds of software vulnerabilities.

In future work, we plan to test our system for larger programs. We also plan to study ways to provide semi-automated methods for identifying other kinds of potential algorithmic-complexity security vulnerabilities rather than simply counting the number of nested loops.

Acknowledgements. This article reports on work supported by the Defense Advanced Research Projects Agency under agreement no. AFRL FA8750-15-2-0092. The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government. This work was also supported in part by the U.S. National Science Foundation under grants 1228639 and 1526631. In addition, we would like to thank David Eppstein, Matthew Might, William Byrd, Michael Adams, and Guannan Wei for helpful discussions regarding the topics of this paper.

References

1. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: COSTA: Design and implementation of a cost and termination analyzer for java bytecode. In: Formal Methods for Components and Objects. pp. 113–132. Springer (2007)
2. Bassi, S., Keller, R.K.: Software visualization tools: Survey and analysis. In: Program Comprehension, 2001. IWPC 2001. Proceedings. 9th International Workshop on. pp. 7–17. IEEE (2001)
3. Bridgeman, S., Tamassia, R.: Difference metrics for interactive orthogonal graph drawing algorithms. In: Whitesides, S.H. (ed.) 6th Int. Symp. on Graph Drawing (GD). pp. 57–71. Springer (1998), http://dx.doi.org/10.1007/3-540-37623-2_5
4. Combinator, U.: Jaam: JVM abstracting abstract machine. <https://github.com/Ucombinator/jaam>, accessed 2016-06-10
5. De Pauw, W., Jensen, E., Mitchell, N., Sevitsky, G., Vlassides, J., Yang, J.: Visualizing the execution of java programs. In: Software Visualization, pp. 151–162. Springer (2002)
6. DeLine, R., Venolia, G., Rowan, K.: Software development with code maps. Commun. ACM 53(8), 48–54 (Aug 2010), <http://doi.acm.org/10.1145/1787234.1787250>
7. Diehl, S., Görg, C.: Graphs, they are changing: Dynamic graph drawing for a sequence of graphs. In: Goodrich, M.T., Kobourov, S.G. (eds.) 10th Int. Symp. on Graph Drawing (GD). pp. 23–31. Springer (2002), http://dx.doi.org/10.1007/3-540-36151-0_3
8. Ellson, J., Gansner, E.R., Koutsofios, E., North, S.C., Woodhull, G.: Graphviz and dynagraph — static and dynamic graph drawing tools. In: Jünger, M., Mutzel, P. (eds.) Graph Drawing Software, pp. 127–148. Springer (2004), http://dx.doi.org/10.1007/978-3-642-18638-7_6
9. Erten, C., Harding, P.J., Kobourov, S.G., Wampler, K., Yee, G.: Graphael: Graph animations with evolving layouts. In: Liotta, G. (ed.) 11th Int. Symp. on Graph Drawing (GD). pp. 98–110. Springer (2004), http://dx.doi.org/10.1007/978-3-540-24595-7_9
10. Fraser, T.: Space/time analysis for cybersecurity (STAC). <http://www.darpa.mil/program/space-time-analysis-for-cybersecurity>, accessed 2016-06-07
11. Gansner, E.R., Koutsofios, E., North, S.C., Vo, G.P.: A technique for drawing directed graphs. Software Engineering, IEEE Transactions on 19(3), 214–230 (1993)
12. Goodrich, M.T., Pszona, P.: Streamed graph drawing and the file maintenance problem. In: Wismath, S., Wolff, A. (eds.) 21st Int. Symp. on Graph Drawing (GD). pp. 256–267. Springer (2013), http://dx.doi.org/10.1007/978-3-319-03841-4_23
13. Havlak, P.: Nesting of reducible and irreducible loops. ACM Transactions on Programming Languages and Systems (TOPLAS) 19(4), 557–567 (1997)
14. van Heesch, D.: Doxygen: Source code documentation generator tool. Available online: <http://www.doxygen.org> (accessed on 8 June 2016) (2008)
15. Heintze, N., McAllester, D.: On the cubic bottleneck in subtyping and flow analysis. In: Logic in Computer Science, 1997. LICS’97. Proceedings., 12th Annual IEEE Symposium on. pp. 342–351. IEEE (1997)
16. Horn, D.V., Mairson, H.G.: Deciding *kcfa* is complete for EXPTIME. In: ACM SIGPLAN international conference on Functional programming, (ICFP’08). pp. 275–282. ACM (2008)

17. Horn, D.V., Might, M.: Abstracting abstract machines: a systematic approach to higher-order program analysis. *Communications of the ACM* 54(9), 101–109 (2011)
18. Might, M.: Environment Analysis of Higher-Order Languages. Ph.D. thesis, Georgia Institute of Technology (2007)
19. Might, M., Smaragdakis, Y., Horn, D.V.: Resolving and exploiting the k -cfa paradox: illuminating functional vs. object-oriented program analysis. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, (PLDI’10). pp. 305–315. ACM (2010)
20. Misue, K., Eades, P., Lai, W., Sugiyama, K.: Layout adjustment and the mental map. *Journal of Visual Languages & Computing* 6(2), 183–210 (1995), <http://www.sciencedirect.com/science/article/pii/S1045926X85710105>
21. Purchase, H.C., Hoggan, E., Görg, C.: How important is the “mental map”? – an empirical investigation of a dynamic graph layout algorithm. In: Kaufmann, M., Wagner, D. (eds.) 14th Int. Symp. on Graph Drawing (GD). pp. 184–195. Springer (2007), http://dx.doi.org/10.1007/978-3-540-70904-6_19
22. Shivers, O.G.: Control-Flow Analysis of Higher-Order Languages. Ph.D. thesis, Carnegie Mellon University (1991)
23. Wei, T., Mao, J., Zou, W., Chen, Y.: A new algorithm for identifying loops in decompilation. In: *Static Analysis*, pp. 170–183. Springer (2007)

A SFR Numbering and Isomorphic Subgraphs

Fig. 7 illustrates a graph (for the complete code of a bubblesort implementation) where the vertices are labeled with their SFR numbers.

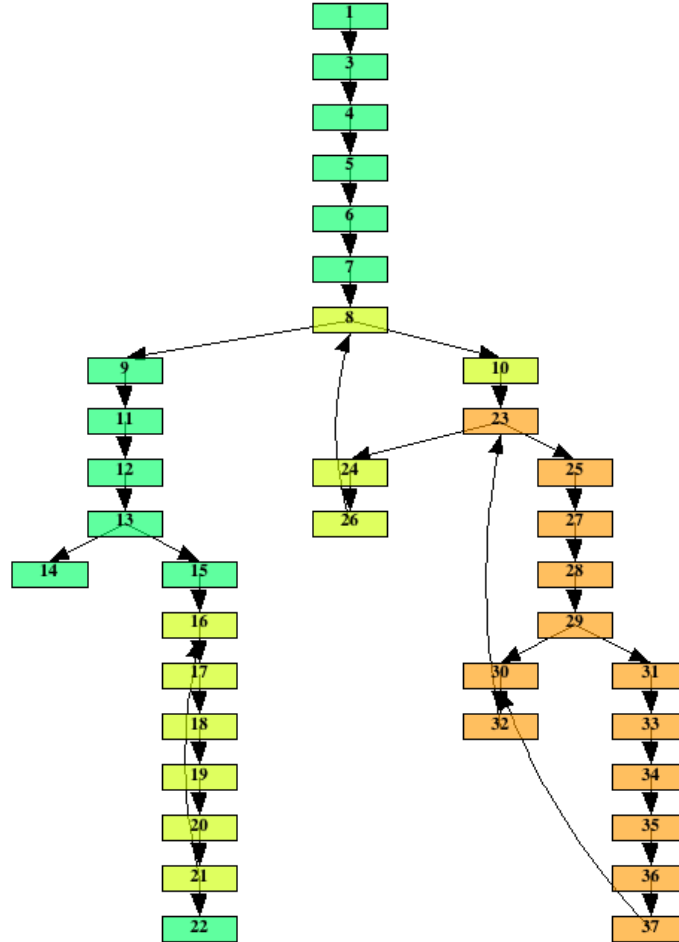


Fig. 7. A graph for the bubblesort algorithm; vertices are labeled with SFR numbers.

Fig. 8 illustrates the difference in the spanning trees and the vertex numberings obtained in our SFR search, and a more conventional depth-first search. Both the trees and vertex numbering is for a code segment containing switch statement. The tree and the numbering obtained in the SFR shows the structure of the switch statement more naturally.

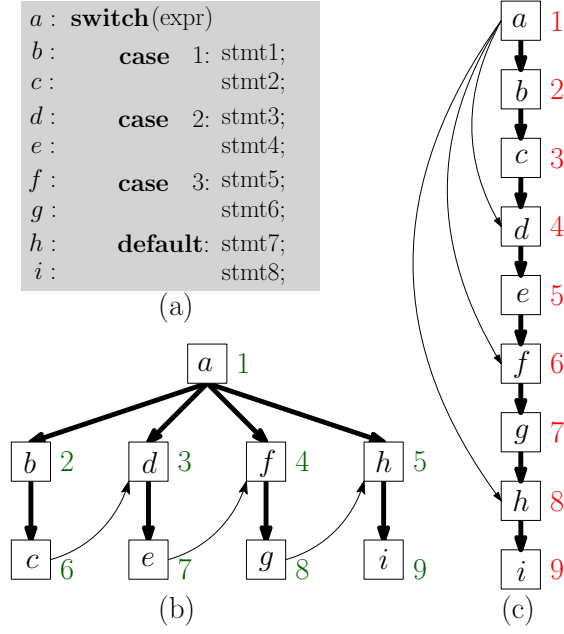


Fig. 8. (a) A code segment containing a switch statement, (b) the spanning tree (thick edges) and vertex numbering (in blue color) for the graph obtained from SFR search, and (c) the spanning tree (thick edges) and vertex numbering (in red color) for the graph obtained from depth-first search.

The SFR search tree and the SFR numbering also enables viewers to visually identify isomorphic subgraphs in the graph, which correspond to identical or equivalent lines of code in the program; see Fig. 9.

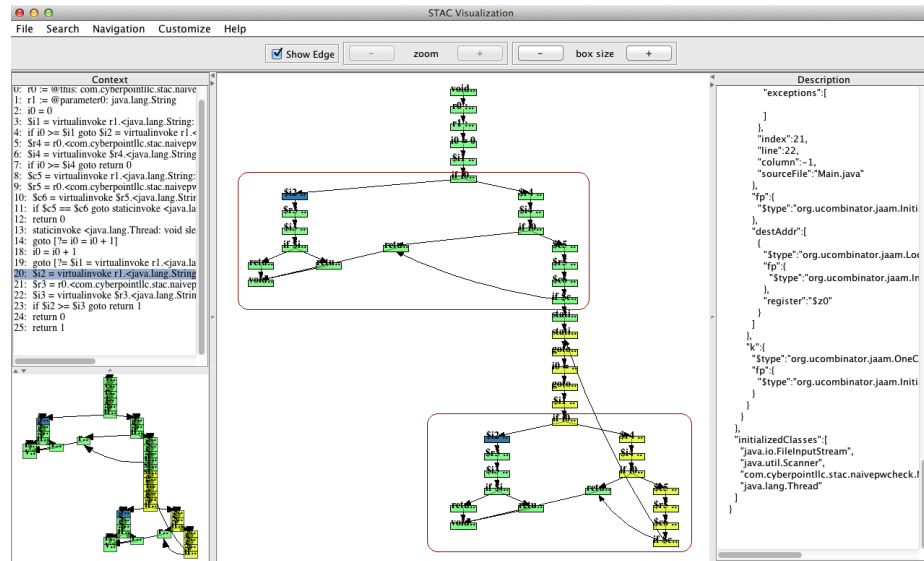


Fig. 9. Visually separable isomorphic subgraphs in the graph for the password checker code; these isomorphic subgraphs correspond to the same or equivalent lines of code.

B Additional Figures and Examples

In this we show the graphs for some simple algorithms, as visualized in our J-viz. In particular we show the graphs for a bubblesort algorithm, for two simple programs implementing the recursive functions computing the factorial of a number and the n -th Fibonacci number, respectively.

Bubblesort Algorithm

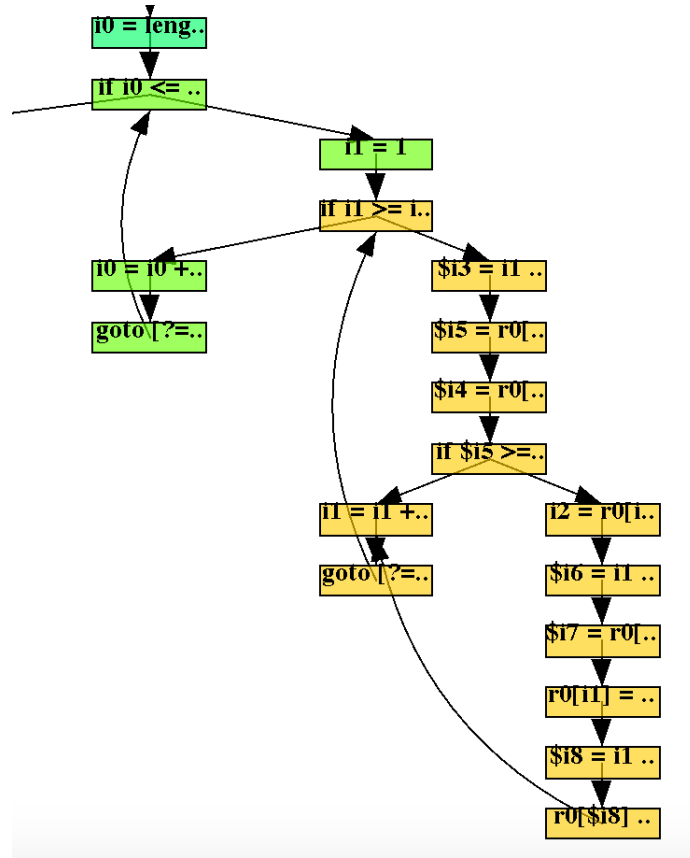


Fig. 10. This partial graph of a bubblesort algorithm shows how drawing upward edges as curves and highlighting nested loops can improve readability. It also shows our use of colors for different levels of nested loops.

Computation of Factorial(n)

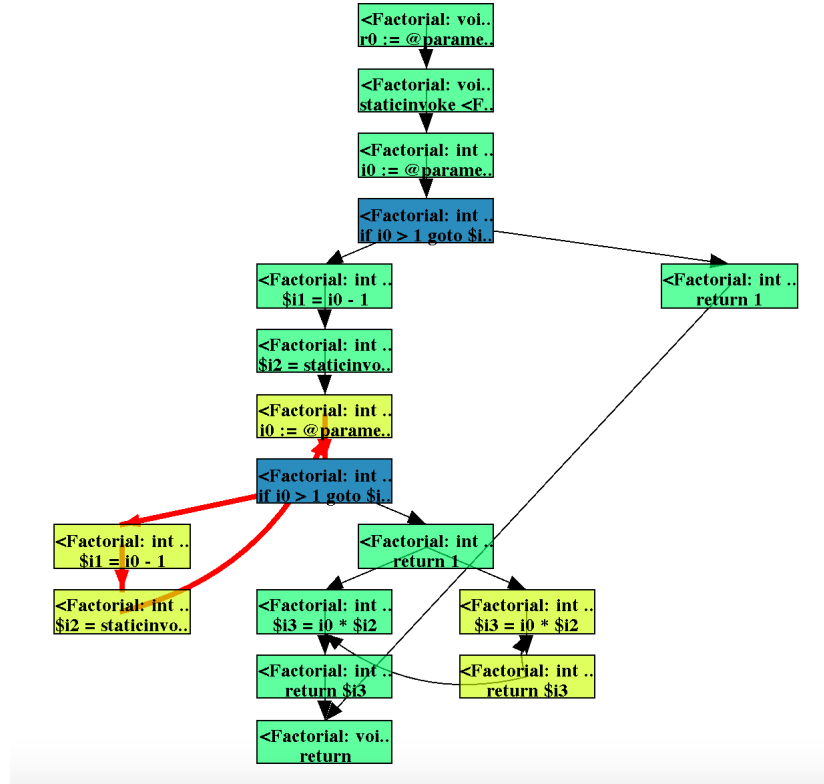


Fig. 11. A graph of a recursive factorial function using 1-CFA. The highlighted instruction occurs twice - once in the context of being called from main, and once in the context of the factorial function calling itself. Note, in addition, how this tree is different from a breadth-first search (BFS) tree. Namely, there is a long forward edge from the rightmost node in the drawing. In a BFS numbering of this graph, that edge would force its end-vertex to a higher level, which would distort the natural notion of recursive depth that the SFR spanning tree illustrates better here.

The control flow graph illustrates a program with the following structure:

- Initial Sequence:** A series of assignment nodes: `PU := ..`, `$r1 := ..`, `$rU := ..`, `$r1 := ..`, and `uU := ..`.
- Loop Header:** An `if uU ..` node that branches to either a `retur..` node or a block of assignment nodes: `$r1 := ..`, `$r2 := ..`, and `uU := ..`.
- Loop Body:** An `if uU ..` node that branches to either a `retur..` node or a block of assignment nodes: `$r1 := ..`, `$r2 := ..`, and `uU := ..`.
- Exit and Return:** The graph concludes with a `virtu..` node and a final `return` node.

Red arrows indicate a specific execution path, starting from the initial sequence, entering the loop, and following the `if uU ..` branches.

20

C Illustration of the J-Viz System

Fig. 13 illustrates our J-viz system and its different component.

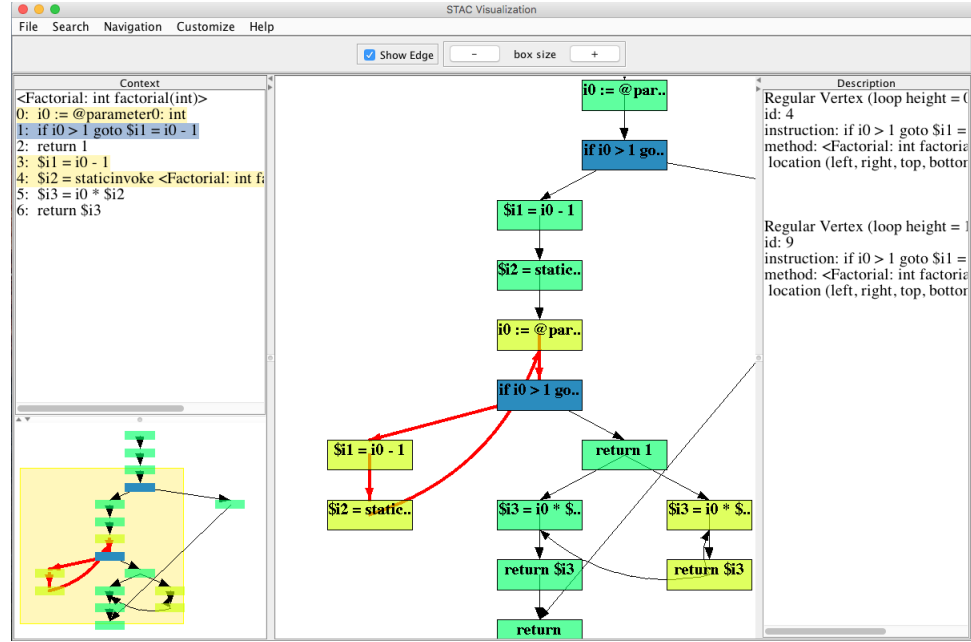


Fig. 13. A screenshot of our complete visualization system, J-Viz, with a (clipped) view of an example Factorial program. This shows the left panel with the code for the currently selected method, the right area with the description of each selected node, and the minimap with our current zoom level.

Fig. 14 illustrates how the J-viz system is used in indentifying vulnerable code segments in a password checker program.

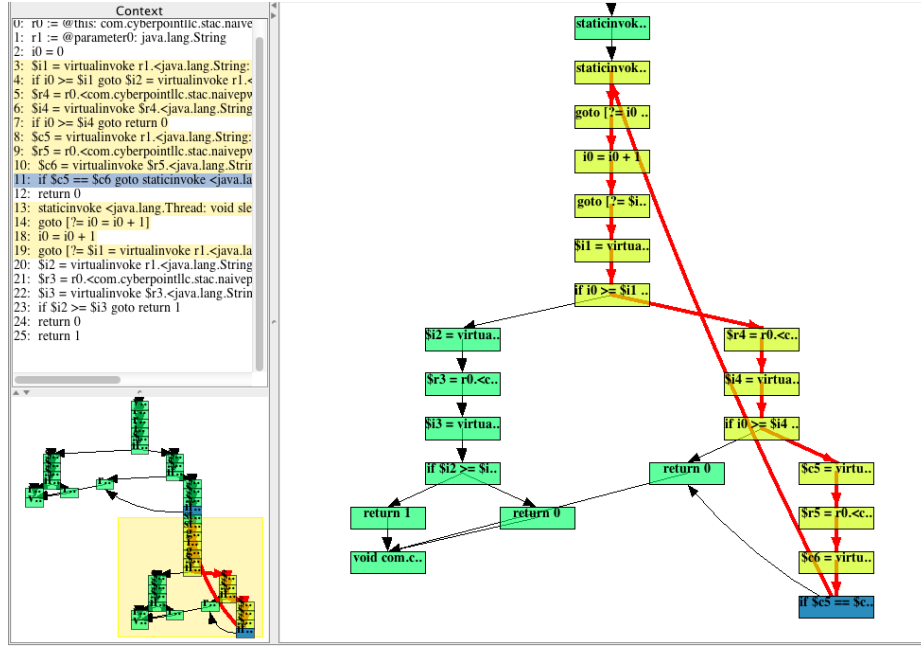


Fig. 14. Our layout of the graph for a password checker with the relevant part zoomed, as a part of our first case study. The highlighted node shows a check for each character of a password. If this fails, the program exits the loop immediately, allowing for a side-channel attack (for identifying failed passwords).